Efficient Barriers for Distributed Shared Memory Computers

Dirk Grunwald Suvas Vajracharya
(grunwald,suvas)@cs.colorado.edu
Department of Computer Science
University of Colorado
Boulder, CO 80309-0430

Abstract

Barrier algorithms are central to the performance of numerous algorithms on scalable, high-performance architectures. Numerous barrier algorithms have been suggested and studied for Non-Uniform Memory Access (NUMA) architectures, but less work has been done for Cache Only Memory Access (COMA) or attraction memory [2] architectures such as the KSR-1.

In this paper, we present two new barrier algorithms that offer the best performance we have recorded on the KSR-1 distributed cache multiprocessor. We discuss the trade-offs and the performance of seven algorithms on two architectures. The new barrier algorithms adapt well to a hierarchical caching memory model and take advantage of parallel communication offered by most multiprocessor interconnection networks. Performance results are shown for a 256-processor KSR-1 and a 20-processor Sequent Symmetry.

1 Introduction

Barriers are a synchronization tool for parallel computers, including shared and distributed (message-passing) address-space architectures. No processor may pass the barrier until all processes have arrived at the barrier; this synchronization tool is used in many algorithms, and is central to the data-parallel programming model [6].

There are numerous barrier algorithms for message-passing and shared-address space computers. Some architectures, such as the Thinking Machines CM-5, provide special hardware support for barrier synchronization. On architectures lacking such hardware support, scalable barriers must be implemented in software, using the underlying communication network. In many ways, the design of scalable barrier algorithms for message-passing machines is easier than than that of shared-address space computers, because communication on such machines is explicitly under programmer control. By comparison, the shared-address space computers rely on caches, prefetching and invalidation to provide the view of a single shared address space, complicating algorithms that rely on exacting analysis of communication patterns.

In this paper, we describe several existing barrier implementations, propose two new barrier algorithm and describe the performance of each algorithm on a Cache Only Memory Access (COMA) architecture, the KSR-1, and a shared-bus cache-consistent architecture (the Sequent Symmetry). We show that conclusions drawn in previous studies of barriers [9] are not necessarily true on COMA architectures, highlighting the need for a formal model to predict the performance of barrier algorithms.

2 Previous Barrier Algorithms

As computer architectures have evolved, numerous barrier algorithms have been proposed. A *central barrier* suffices for small-scale cache-coherent multiprocessors with 2-10 processors. As the promise of larger-scale systems including tens and hundreds of processors was realized, it was noted that centralized barriers limited system performance. Concentrated communication to a single memory location can induce a *hot spot* [10] in the network. Both hardware solutions, including combining networks, and software solutions [12] have been proposed to alleviate hotspots. Several barrier algorithms have been described that distribute the communication, either over different cache locations, network connections or processor clusters. Most of these algorithms use some tree-based structure to distribute the communication. In the remainder of this section we describe several such algorithms.

We assume a barrier algorithm has three important components. The data structure for the barrier must created – many barrier algorithms use a data structure (typically a $P \times \log P$ matrix for P processors) to guide the processors during arrival. This matrix is created when the barrier is created, and is typically reused during each barrier rendezvous.

Each rendezvous (i.e. each time the barrier is used) consists of an *arrival*, *wakeup notification* and *reinitialization*. Processors must inform each other that they have arrived at the barrier; when all processors have arrived, processors must be alerted (wakeup) so they can continue execution. Once the barrier has been reached, the barrier is reinitialized so that it can be reused; note that this does not involve recomputing the information generated when the barrier was created. Algorithms differ in the design and use of arrival, wakeup and reinitialization mechanisms.

2.1 Central Barrier

The most obvious implementation of a barrier is an accumulating counter. The counter is initialized to zero. Processors arrive at the barrier by atomically incrementing the counter; they receive wakeup notification by waiting until until the counter reaches the number of expected processors. The waiting is usually implemented by busy-waiting, or "spinning". Once the rendezvous has been achieved, the counter is reset to zero for the next rendezvous.

There are two disadvantages to this approach. First, the counter must be updated atomically, either via explicit locking or hardware operations such as `fetch_and_`ϕ [3]. Second, all processes must contend with each other to read and write a single memory location. As mentioned, this causes hot-spots, or points of high traffic congestion. Consequently, this barrier is not scalable since each read and a write involves serialized actions. The hotspot problem can be somewhat alleviated in cache coherent machines because a copy of the counter is copied to local memory and updated *via* broadcasts; this reduces the amount of network communication. Various centralized barrier and lock designs make effective use of local caches [1]. However, processors writeing to the barrier contend between themselves, and arrival still requires O(N) time.

2.2 Software Combining Trees

Yew *et al.* [12] proposed a combining tree barrier to reduce the occurence of hot spots. A software combining tree spreads the congestion over a tree of variables. Arriving processors are divided into pre-determined groups. During a rendezvous, the processor arriving last in a given group becomes the group representative and advances to next level in the tree. The group representatives now wait for other group representatives, forming a new group at the next level. This procedure is repeated until one processor reaches the root. Software barriers are initialized by creating the tree data structure and initializing each node with the count of processors expected to report to that node. Processors that are not group leaders busy-wait, awaiting wakeup notification of the barrier completion.

A number of methods can be used for wakeup notification. In broadcast cache-coherent architectures (such as the KSR or Sequent), a central busy-flag can be used [9]. On architectures where non-local memory references must use the network, a tree-based wake-up algorithm is used. The counters at each node are reset when the barrier is reinitialized.

This presents a design choice exercised in different ways by the various barrier algorithms – decreasing hot spot contention *vs.* increasing the critical path for arrival and wakeup notification. In the simple accumulating counter, the critical path is one operation; however, P writers must contend for a serialized resource. At the expense of increasing the critical path to $\log P$, the various combining tree algorithms increase the amount of parallelism and therefore the number of (mild) hot spots. We will later discuss a method to increase the parallelism in the barrier algorithm; this increases the number of "hot-spots" in the interconnection network, but reduces the contention at each hot-spot.

2.3 Dissemination Algorithm

The Combining Tree barrier shares two disadvantages with the simple accumulating counter. First, both require atomic operations when updating the counters, such as compare and swap, locking or `fetch_and_`ϕ. Second, both methods spin at a remote memory location, leading to unnecessary contention for the interconnection bandwidth on machines that are not broadcast-based and lack cache-coherency.

To alleviate these problems, Brooks described the *butterfly barrier* [7]; this communication pattern is similar to the *exchange-swap* operation used in hypercube interconnection networks. Hensgen *et al* [5] improved the butterfly-network for situations where the number of processors accessing the barrier are not a power of two. Their *dissemination barrier* uses the communication structure shown in Figure 1.

Unlike the software combining tree barrier, the rôle of each processor at each round is staticly pre-determined, and there is no need for atomic locking or `fetch_and_`ϕ instructions. The barrier is created for P processors by creating a $P \times \log_2 P$ matrix containing flags and pre-determining the processes that should receive (wait for the flag) and the processes that should send (set the flag) at each of the $log_2 P$ rounds. In the dissemination barrier, arrival and wakeup notification are combined into $\log_2 P$ rounds for P processors. For round k, processor i signals its "partner processor" ($j = i + 2^k \bmod P$) thereby indicating the arrival of processor i and all processors that have synchronized with processor i in previous rounds. During each synchronization 'round', each processor either waits on a locally available flag or signals the partner *via* writing the flag. The flags are reinitialized by sense-reversing the flag after each round.

In a previous study [9], the dissemination algorithm achieved the best performance of a variety of barrier algorithms on non-uniform memory shared-address machines without cache-coherency and broadcasting, such as BBN Butterfly. On such architectures, the 'local spinning' provides an advantage for this algorithm. This is a less of a factor in COMA architectures, since there are no fixed homes for any memory locations. Without the advantage of the local spinning, but with the disadvantage of incurring $O(P \log_2 P)$ distinct network transactions, this algorithm does not perform as well as others on COMA architectures.

The dissemination algorithm does not adapt well to hierarchical memory model machines such as the KSR-1, where the cost of communication is not constant between all processors. In these systems, processors are partitioned into *clusters* of processors (e.g., into distinct rings on the KSR). Figure 1 demonstrates that there is no way to map the virtual processors to physical processors while reducing inter-cluster communication. We believe the communication structure of a barrier algorithm must match the physical interconnection network for best performance.

2.4 Tournament Algorithm

To counter the additional communication in the dissemination algorithm, Hensgen *et al* also developed the tournament algorithm, apparently at the suggestion of Lubachevsky [8]. As with the dissemination algorithm, the tournament algorithm avoids special hardware by using a pre-determined communication structure; however, the tournament algorithm incurs only $O(\log_2 P)$ total network transactions, as illustrated by the communication structure shown in Figure 1. As with the dissemination algorithm, communication is broken into $\log P$ rounds, such that at round k processor i sets the flags for itself and it's 'partner processor', processor $j = i - 2^k$.

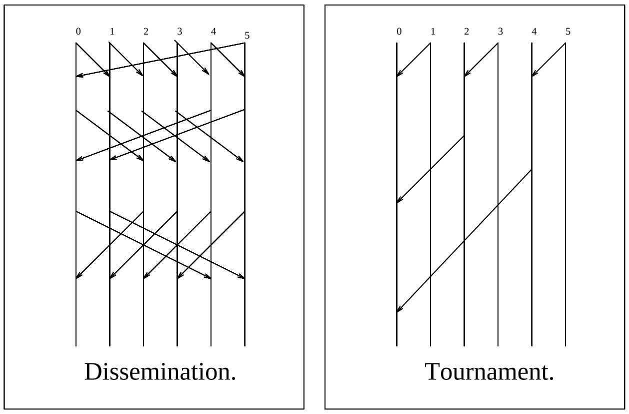

Figure 1: Communication Structure for Tournament and the Dissemination Algorithms

Arrival notification involves $\log_2 P$ communication rounds for P processors, where "winning" processors wait for "losing" processors and "losing" processors set the flag for the "winning processors." The "winners" and "losers" are determined during barrier creation, and each processors rôle is stored in a $P \times \log_2 P$ matrix The "losers" do not participate in the remaining rounds and wait for the wakeup notification. Wakeup notification is done using either a global wakeup flag for broadcast-based machines or a tree-wakeup algorithm. Reinitialization is done by sense reversing the flags.

2.5 MCS-Tree Algorithm

Mellor-Crummey *et al* [9] proposed a variation on the tournament algorithm called the MCS-Tree barrier. Figure 2 depicts the communication structure for the MCS-Tree barrier algorithm, and Figure 3 presents an alternate view for a larger number of processors. In this algorithm readers (parents) and writers(children) are statically pre-determined during barrier creation.

When a child arrives at the barrier, it writes to the parents memory; parents spin until all children have arrived. The flags written by the children are packed into a single word, reducing the time for the parent to check that all children have arrived. During barrier creation, processor i selects processor $\lfloor \frac{i-1}{4} \rfloor$ as its parent, and will write to byte $(i-1) \bmod 4$ of the parents flag. The four-fold fan-in was selected because Yew *et al* [12] reported this resulted in the best speedup in their software combining tree and four bytes are

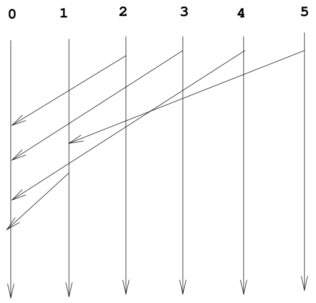

Figure 2: Communication Structure Used in MCS-tree Algorithm

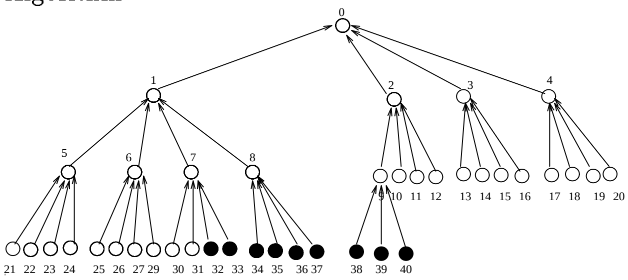

Figure 3: Example of MCS Barrier With More Processors

easily packed into a 32-bit word. During notification, each processor spins on its flag-word until all bytes are cleared by the arriving children and then informs its own parent. Again, wakeup notification is done using either a global wakeup flag for broadcast-based machines or a tree-wakeup algorithm. Reinitialization is done by sense reversing.

3 New Barrier Algorithms

3.1 Static f-Way Tournament

The Static f-Way tournament algorithm is, as implied by the name, a variation of the tournament algorithm using a f-way fanin rather than the 2-way fanin in the original tournament algorithm. Figure 4 illustrates the communication structure for the Static f-way Tournament algorithm when f is 4. The actual value for f is determined by the processor architecture and the number and physical placement of processors participating in a given stage of the barrier, and may vary across different levels of the synchronization tree. The fanin, f, is varied such that each level of the synchronization tree is balanced as much as possible while reducing inter-cluster references. This reduces the critical path length for this algorithm to

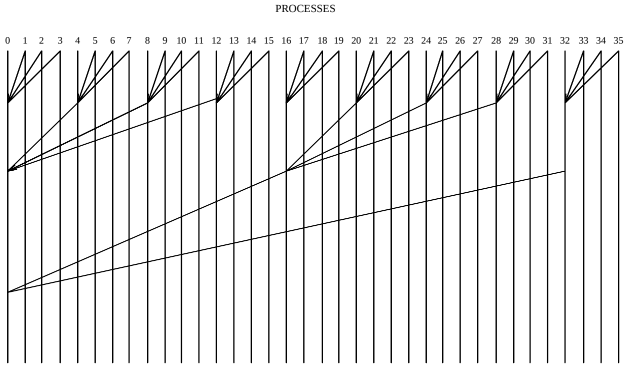

Figure 4: Communication Structure Used in the f-way Tournament Algorithm

$\log_f P$ for P processors. The parameter f varies between two and the number of bytes in memory word. Due to space considerations, we can not show the full algorithm in this paper; interested readers are referred to an extended version of this paper [4] for more information. Furthermore, implementations of these algorithms are available; contact the authors for more information.

As with other algorithms, a $P \times \log_f P$ matrix is used to hold pre-determined scheduling information about the rôle of each processor in the barrier. As with the prior algorithms, child processors contact their parents. Due to the pre-determined schedule, some processors may not participate in a particular round. Parents continue execution, becoming a parent again or turning into a child. Children await the wakeup notification; we use a global variable for wakeup, although a tree-based wakeup could also be used.

Barrier creation and initialization of the data structures is more complicated than in other algorithms we have examined. We use a function, `fanin`, shown in Figure 5, that computes the approriate fanin based on the current fanin and number of remaining processes to synchronize at any level in the tree. The choice of fanins and this function will be discussed fully later.

At each level of the tree, processors record the information needed to contact their parent processor in that round of synchronization. In practice, the algorithm uses pointers to shared data structures for efficiency. As in the MCS-Tree barrier algorithm, we use a packed data structure to simplify checking for the arrival of children.

3.2 Dynamic f-Way

The Static f-way Tournament, MCS-tree and the Tournament algorithms are all *static* – the rôle of each

```
procedure fanin(num_procs_left : in Integer)
   returns Integer
begin
  best_fanin := MAX_FANIN;
  best_measure := 0;
  while (current_fanin > MIN_FANIN) do
      remainder := num_procs_left
                   mod current_fanin;
      if (remainder = 0) then
        remainder := current_fanin;
      end if;
      measure = remainder div current_fanin;
      if (measure > best_measure) then
          best_measure := measure;
          best_fanin := current_fanin;
      end if;
      current_fanin := current_fanin - 1;
  end while;
  if (MIN_FANIN > 2)
     MIN_FANIN = MIN_FANIN -1;
     MAX_FANIN = MAX_FANIN -1;
  end if;
  return best_fanin;
end fanin;
```

Figure 5: Computing the Fanin For the f-way Barriers

processor is predetermined during barrier creation.

There are two weaknesses with such static algorithms: the latency in detecting the arrival of all the children in a group is slightly longer, and these algorithms require that the pre-determined parent poll the arrival of the pre-determined children. On COMA memory systems such as the KSR-1, or distributed cache consistent architectures, this may cause many unnecessary cache faults. These architectures typically use an invalidation write-allocate policy for cache replacement with write-back updating. In the case of the KSR, there is no updating, since all memory in the system acts like a cache. A memory write by one processor invalidates references on all other processors and the item is allocated on the writing processor.

Since the order of the arrival of processes at a barrier is non-deterministic, it is possible for the parent to arrive before its children. The children (writers) arriving after the parent(reader) will suffer a cache miss to bring the shared-line to their local memory. At this point, the parent polling this invalidated cache-line has another cache fault, drawing the word back to the local memory. Because of this effect, the expected number of cache faults at an internal synchronization point in the Static f-way Tournament is not $f-1$, the theoretical minimum, but $\frac{3(f-2)}{2}$.

For example in the Tournament algorithm, where f is fixed at 2, the expected number of cache faults is $\frac{3}{2}$. For MCS-Tree, we expect the number of cache faults at an internal synchronization point to be no better than $\frac{3f+1}{2}$, because the parents in this algorithm start polling at the internal synchronization points immediately after entering the barrier, instead of starting at the leaves and climbing up the tree. This increases the chance of the parent arriving before its children, exacerbating the cache invalidations.

If the synchronization point at each internal node involves processes from different rings, these cache faults can be very expensive; for example, communication within a ring on the KSR-1 takes ≈ 150 cycles, while communication between rings requires ≈ 600 cycles. Other architectures, such as the Convex SPP have similar access latencies. Dynamic algorithms such as the Software Combining tree and the Central algorithm avoid this problem by not mixing polling and writing to the same memory location. Not counting the cache fault incurred to acquire the lock, the worst case is only f writes, the theoretical minimum. As mentioned earlier however, the Software Combining algorithm, as originally proposed, suffers from two problems: it requires special hardware and it does not adapt well to hierarchical memory models.

Recognizing these problems, we now propose a new algorithm called the *Dynamic f-way*. This algorithm has some of the same benefits as the Static f-way Tournament, such as adaptability to hierarchical memory model, freedom from special hardware instructions, and a high-degree of parallelism – yet does not induce as many cache faults. Again, the full algorithm is described in the longer version of this paper [4].

Like the Software Combining Tree, the last process to arrive at a node goes on to participate at higher level node(s). Like the f-Way Tournament, communication between neighboring processors (again assuming that virtual process id i is mapped to physical process i) is done before communication with distant processes – in fact the communication structure is the same. Also, as in the f-way Tournament, a variable fanin maybe used each process writes to one dedicated byte in a word of to allow its parent to efficiently note the childs arrival.

This last analogy is not quite true. We actually use a sense-reversing mechanism to note the arrival of children. Using a simple count introduces a race condition into the algorithm. To show why this is necessary, consider the algorithm without the use of the reversal (as in the MCS-tree and the Static f-way Tournament). Table 1 shows the race condition that can occur at an internal node of the tree. The columns labeled `expected` and `so_far` are words that contain

Time	**Process Id**	**Rendezvous**	**Event**	**Expected**	**So_Far**
0	–	–	Initial	[1,1,1,0]	[1,1,1,0]
1	I	K	Clear Flag	[1,1,1,0]	[0,1,1,0]
2	I+1	K	Clear Flag	[1,1,1,0]	[0,0,1,0]
3	I+2	K	Clear Flag	[1,1,1,0]	[0,0,0,0]
4	I+2	K	Copy Flag Word	[1,1,1,0]	[1,1,1,0]
5	I+1	K	Achieve Barrier	[1,1,1,0]	[1,1,1,0]
6	I	K	Achieve Barrier	[1,1,1,0]	[1,1,1,0]
7	I	K+1	Clear Flag	[1,1,1,0]	[0,1,1,0]
8	I+1	K+1	Clear Flag	[1,1,1,0]	[0,0,1,0]
9	I+1	K	Copy Flag Word	[1,1,1,0]	[1,1,1,0]

Table 1: Race Condition in the Incorrect Algorithm

	Central	Combining	Dissem.	Tourn.	MCS-Tree	Stat. f-way	Dyn. f-way
Special Hardware	Yes	Yes	No	No	No	No	No
Critical Path	1	$\lceil \log_f P \rceil$	$\lceil \log_2 P \rceil$	$\lceil \log_2 P \rceil$	$\lceil \log_f P \rceil$	$\lceil \log_f P \rceil$	$\lceil \log_f P \rceil$
Network Transactions	P-1	$\lceil \frac{P-1}{f-1} f \rceil$	$P \lceil \log_2 P \rceil$	P-1	P-1	P-1	$\lceil \frac{P-1}{f-1} f \rceil$
Network Transac.(COMA)	P-1	$\lceil \frac{P-1}{f-1} f \rceil$	$2P \lceil \log_2 P \rceil$	$2(P-1)$	$\lceil \frac{P-1}{f-1} \frac{3f+1}{2} \rceil$	$\lceil \frac{P-1}{f-1} \frac{3f-2}{2} \rceil$	$\lceil \frac{P-1}{f-1} f \rceil$
Number of Writers	P	f	1	1	f	f-1	f
Hierarchical Memory	No	No	No	Yes	No	Yes	Yes
Communication Schedule	Dynamic	Dynamic	Static	Static	Static	Static	Dynamic
Space	1	P	$P \lceil log_2 P \rceil$	$\frac{P}{2} \lceil log_2 P \rceil$	P	$\frac{P}{f} \lceil log_f P \rceil$	$\frac{P}{f} \lceil log_f P \rceil$

Table 2: A comparison of barrier algorithms. For the Static f-way Tournament and Dynamic f-way, the fanin may vary across different levels of the tree. We have assumed a fixed fanin for the table above in order to compare these two algorithm with the others

the information on what children are expected and what children have arrived at a particular node of the synchronization tree. The actions of the processors spans two barrier rendezvous', denoted as K and $K+1$. At time step 9, process $I+1$ is is still in rendezvous K. Process $I+1$ re-copies the the word flag, corrupting the information about the arrival of processes I and $I+2$, both of which are in epoch $K+1$.

Even with the reversal of the '`expected`' field in the correct algorithm, it is still possible that more than one child acts as the parent of any given branch and continues to climb up the tree. However, this is not a problem because any two consecutive epochs of the barrier-rendezvous run in two distinct flag spaces i.e. two separate trees of flags.

3.3 Comparison of Expected Performance

A comparison of the expected performance of this algorithm with the previously discussed algorithms is summarized in Table 2. For the Static f-way Tournament and Dynamic f-way, the fanin may vary across different levels of the tree. In order to compare these two algorithms with the others, we used a fixed fanin in Table 2. The first row indicates whether special hardware instructions other than atomic read and write are required. The second list the expected critical path length. The third and fourth rows of network transactions for each barrier algorithm, the later being the *expected* number of network transactions on COMA machines. In general, the total number of network transactions can be computed as the product of the number of internal synchronization points in the tree $(\frac{P-1}{f-1})$ and the number network transactions at each internal synchronization point. The latter is simply the number of writers at an internal synchronization point on non-COMA architectures. On COMA architectures, the number of network transactions at a node depends on the order of the arrival of writers and readers: the fourth row shows the *probabilistic* estimate of the total number of transactions. For example, the number of network transactions at an internal node for the Static f-way Tournament is $(f-1)+1$ in the best case but $2(f-1)$ in the worst case, giving an average of $\frac{3f-2}{2}$. Hence we expect $\lceil \frac{P-1}{f-1}\frac{3f+1}{2} \rceil$ network transactions for this algorithm. The fifth and the sixth rows indicate the contention at a "hot spot" and type of scheduling used by that algorithm. The seventh row indicates whether the algorithms are well suited to hierarchical memory architectures, i.e. whether there is a simple mapping between virtual and physical processors such that communications between distant processors are minimized. Finally the last row shows the shared-memory space required by the each algorithm.

The fourth and fifth row suggest that the static algorithms (Dissemination, Tournament, MCS-Tree, Static f-way Tournament) incur more network transactions and cache faults on COMA architectures than in other architectures. This implies that dynamic algorithm will have an advantage on the KSR-1. Among the dynamic algorithms, Table 2 indicates that Dynamic f-way improves upon both the Central and Software Combining Tree. Among the static algorithms, Static f-way Tournament improves upon Dissemination, Tournament, and the algorithm previously known to be best for cache-coherent machines, the MCS-Tree. We note that Tournament is identical to Static f-way Tournament with f fixed to 2. Static f-way Tournament improves on MCS-Tree because there are fewer writers for any given memory location, resulting in less contention for that memory location. For P processors and a critical path of $\log_f P$, only $f-1$ writers and 1 reader contend for a given memory location.

For example, given 125 processors, the f-way tournament can achieve a rendezvous with 4 writers per memory location and a critical path of 3. By comparison, the MCS algorithm could only synchronize 64 processors using the same path length and number of writers. The Static f-way Tournament algorithm builds the tree from bottom up, in contrast to the top down tree construction in MCS-Tree. This means that the tree in Static f-way Tournament is balanced if the number of processors (the leaves) are a multiple of the fanin. The fanin may be varied at different levels of the tree, leading to more balanced tree. In the MCS-tree, a balanced tree is only achieved if number of processors is a power of the fanin. Balanced trees lead to a greater degree of parallelism, which can be adequately supported by machines like the KSR-1.

4 Choice of fanin

The fanin has has a significant impact on the speed of tree algorithms. If the fanin is small, then the critical path from leaf to root is increased. If the fanin is large, however, more processors must write to a single memory location, increasing the contention for a particular synchronization point.

We can approximate the desired fan-in using the following simplifying assumptions:

1. The number of processors, P is power of the fanin, f.

2. Processors (writers) arriving at an internal node must report their arrival sequentially i.e. if there are f writers and each write takes c time i.e. the total time is fc. This assumption is not true for example if we use hardware implemented *Fetch & Add* as means of synchronization at an internal node.

3. The time required to execute the instructions for climbing the tree after the arrival of processes in a group is not significant.

The total time to complete a single rendezvous is the critical path, $\log_f P$, times the time required at each internal node, which we have assumed to be fc. We can then solve for f and minimize the total time by taking the derivative of this expression with respect to the fanin.

$$
\begin{aligned}
T(f) &= fc\log_f P = fc\frac{\ln P}{\ln f} \\
T'(f) &= c(f(0 - \ln P(\frac{1}{f})) + \frac{\ln P}{\ln f}) = c(\frac{\ln P}{\ln f} - \ln P)
\end{aligned}
$$

As in most 'optimal' tree algorithms, $T'(f)$ is zero when $f = e$, or approximately 3. Because our assumptions are not always true, and more importantly, because the communication networks in many architectures do not adequately support the degree of parallelism offered by a lower fanin, the optimal fanin in many machines is closer to 4. Also as shown in Table 2, a larger fanin also reduces the total number of network transactions on COMA architectures.

The right choice of fanin is further complicated by the number of inter-cluster communications in hierarchical memory architectures, which is a function of the fanin. A larger fanin, particularly at higher levels of the tree, would force many inter-cluster communications. Yet another factor influencing the choice fanin is the balance of the tree. Figure 5 shows a way of choosing fanin for Static f-way Tournament and Dynamic f-way that we used on the KSR-1.

5 Experimental Results

Our implementation of the various barriers discussed in this paper on KSR-1, Sequent Symmetry and BBN-TC2000 is available in the publicly accessible ftp site, `cs.colorado.edu` in the directory `/pub/distribs/Awesime/barriers`.

5.1 KSR-1

Figure 7 shows the timing measurements comparing six previous algorithms and the two new ones. Results are average of 10 separate runs, each run consisting of 30,000 barrier redenvous' with an empty critical section. For all the tree-based algorithms, we used a central global flag that is broadcasted to the waiting processes during the wakeup phase of the algorithms. This method was shown to be more appropriate in a previous study [9] for cache-coherent machines that provide fast broadcast operations.

On the KSR, we implemented the broadcast using the `_pstsp` (post-store subpage) instruction. In theory, we can use this instruction to update the contents of the modified word in a single tour of the network. If we did not use this mechanism, we would invalidate those copies, possibly causing caches misses on those processors.

However, we found that this mechanism does not significantly improve the speed on the Dynamic f-way and Static f-way Tournament algorithms. Because we wanted to keep these algorithm free of special hardware instructions, we have abstained from using `pstp`. For the algorithm that require locks (Software Combining Tree and Central Barrier), we used *acquire-subpage* and *release-subpage* operations on the KSR. There are two versions of *acquire-subpage* on the KSR-1: one that waits until the lock is free and the other that simply returns failure if the lock belongs to another process. We used the latter with polling, as recommended by KSR.

For all the algorithms we bind virtual process i to physical processor i such that consecutive virtual processor numbers map to proximate physical processors. This mapping minimizes cross-ring communication for the Tournament, Static f-way Tournament, Dynamic f-way algorithms and to a lesser degree for other algorithms. By default, the KSR-1 maps consecutive virtual processors to contiguous physical processors when there are sufficient processors available in the system.

Better mappings from virtual to physical processors exist for Software Combining Tree and MCS-Tree for a given number of processors participating at the barrier. As shown in Figure 3, in the MCS-Tree algorithm, child i communicates with its parent, processor $\lfloor \frac{i-1}{4} \rfloor$. This implies that the distance between the child and parent increases with the number of processors. For example, on a 64-processor KSR machine, each of the blackened nodes shown in the barrier in Figure 3 must communicate across processor clusters. However, the general algorithm for the computing this mapping is complex and moreover, the default map-

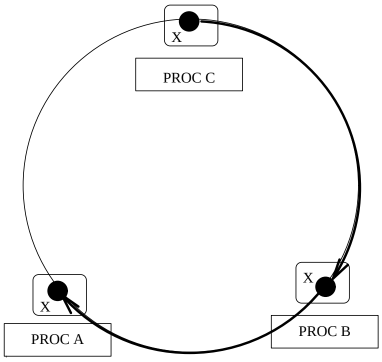

Figure 6: Validation of a cache line on the KSR-1

ping by the KSR-1 operating system does not favor these two algorithms.

Figure 7 shows our empirical results, measured on a 256-processor KSR-1; these results confirm the expected performance. The "system" algorithm is the native barrier implementation provided with the KSR Pthreads library; we do not know the underlying algorithm. Note that the dissemination barrier suffers a sharp increase in the time to rendezvous between 32 and 33 processors; this is when inter-cluster communication occurs in the KSR-1 interconnection network.

Previous studies [9] have shown that the MCS algorithm was faster than other algorithms such as the tournament algorithm. However, there is a greater degree of parallelism in the tournament algorithm, and that parallelism can be supported on the KSR-1. This was also recently noted by Ramachandran *et al* [11]. Furthermore, the tournament algorithm involves less inter-cluster communication, which is very important on the KSR-1 – intra-cluster memory references take 150 machine cycles, while inter-cluster references take ≈ 600 cycles.

Recall that we speculated that 'dynamic' algorithms incurr fewer cache faults than 'static' algorithms. incurred by individual processors, and Figure 8 shows the total number of cache faults incurred after 30,000 iterations of some of the algorithms. We used the pmon performance monitoring hardware provided on each processor of the KSR-1 to obtain these measurements. As expected, the dynamic algorithms (Software Combining Tree and Dynamic f-way) incurred fewer cache faults than the static algorithms. It is worth noting that the number of cache faults is different from the number of network transactions as calculated in Table 2. The former can be a much smaller number as shown in Figure 6. Processor A validates memory location X living in processor C by pulling it to its' local cache and also invalidates the cache line

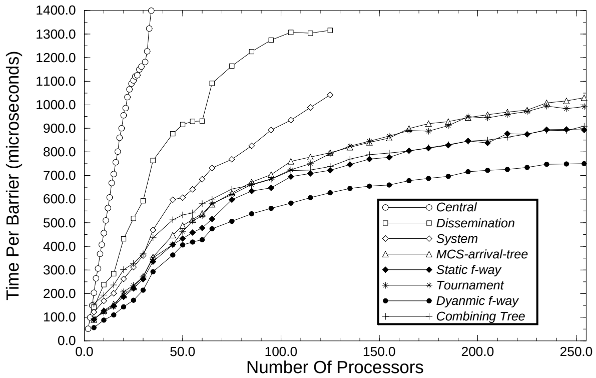

Figure 7: Comparison of Different Barrier Algorithms On 256-Processor KSR-1. Each Algorithm Uses a Broadcast Wakeup Rather Than Tree Wakeup

in processor B. In the diagram, there are two transactions in the diagram but only A suffers the cache fault.

5.2 Sequent Symmetry

As noted by Mellor-Crummey *et al*, the central barrier algorithm out-performs other algorithms on broadcast bus-based architectures such as the Sequent Symmetry for a reasonable ($\leq 16-18$) number of processors. In [4], we present data that verifies their results, and show that the Static f-way Tournament is slightly faster than the other algorithms for a larger number of processors. Since the Dynamic f-way requires more network transactions for non-COMA architectures like the Sequent Symmetry, we do not expect its performance to be competitive with Static f-way Tournament, and have not included this algorithm in the results.

6 Conclusions & Future Work

We have demonstrated, using both static analysis and empirical measurements that our variants on the Tournament algorithm, the Static f-way and the Dynamic f-way algorithm, out-perform currently proposed algorithms. We also showed that for COMA machines such as the KSR-1, the Dynamic f-way algorithm is superior to Static f-way. For other cache

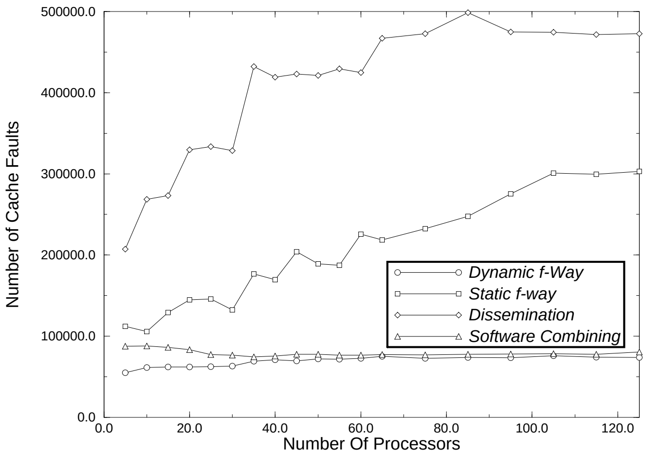

Figure 8: Total Number of Cache Faults For 30000 Iterations of Barrier Rendezvous

coherent machines with 'fixed' memory locations (i.e., non-COMA architectures), we recommend the Static f-way algorithm, which was shown to perform the best in Sequent Symmetry.

We are currently expanding on the algorithms presented here; currently, we have found that the Dynamic f-way algorithm is faster than the other algorithms on the BBN TC-2000; see [4] for more details. We are also considering the interaction of the barrier algorithms with operating system rescheduling; this will necessitate the recomputation of the predetermined information associated with the different processors.

This work was funded in part by NSF grant No. ASC-9217394 and used resources provided by the National Consortium for High Performance Computing (NCHPC) located at the University of Colorado and the Army Research Lab. We would like to thank Harry Jordan, Gary Nutt, Michael Scott, and Dave Wagner for comments on earlier versions of this paper.

References

[1] T.E. Anderson. The performance of spinlock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[2] Seif Haridi Erik Hagersten and David H.D. Warren. Cache and interconnect architectures in multiprocessors. *The cache-coherence protocol of the data diffusion machine*, 1990.

[3] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larray Rudolph, and Marc Snir. The nyu ultracomputer: Designing a mimd, shared-memory parallel machine. *Proceedings of 9th Annual International Symposium on Computer*, 10(3):27–42, April 1982.

[4] Dirk Grunwald and Suvas Vajracharya. Efficient barriers for shared memory computers. CU-CS 703-94., Univ. of Colorado, Boulder, Campus Box 430, Univ. of Colorado, Boulder CO, January 1994.

[5] D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *Intl. Journal of Parallel Programming*, 17(1), 1988.

[6] W. Daniel Hillis and G.L. Steele. Data parallel algorithms. *Communications of the ACM*, 29, No.12:1170–1183, December 1986.

[7] Edward D. Brooks III. The butterfly barrier. *Intl. Journal of Parallel Programming*, 15(4):295–307, 1986.

[8] Boris D. Lubachecsky. Synchronization barrier and relation tools for shared memory parallel programs. In *Proc. of the 1989 Int. Conf. on Parallel Processing*, pages II–175–II–179. Penn State, 1989.

[9] John Mellor-Crummey and Michael Scott. Algorithms for scalable synchronization on shared memory multiprocessors. *ACM Trans. on Computer Systems*, 9(1):21–65, February 1992.

[10] G. Pfister and V. Norton. Hot spot contention and combining in multistage interconnection networks. *IEEE Transactions on Computers*, C-34(10):943–948, October 1985.

[11] Umakishore Ramachandran, Gautam Shah, S. Ravikumar, and Jeyakumar Muthukumarasamy. Scalability study of the ksr-1. GIT-CC 93/03, Georgia Inst. of Technology, 1993.

[12] Pen Yew, N. Tzeng, and Ducan Lawrie. Distributing host-spot addressing in large-scale multiprocessors. *IEEE Transactions on Computers*, pages 388–395, April 1987.